

MA 3046 - Matrix Analysis
Laboratory Number 7
LU Decomposition and Numerical Accuracy

Gaussian elimination, in its most “vanilla” form, i.e. without any row interchange, provides a simple algorithm, perhaps even deceptively so, for solving the system of linear equations

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad , \quad \mathbf{A} \in \mathbb{C}^{m \times m} \quad (1)$$

We simply use elementary row operations of the form:

$$R_i \leftarrow R_i - \ell_{ik} R_k \quad , \quad i > k \quad (2)$$

to sequentially reduce the $m \times (n+1)$ augmented matrix to echelon (augmented) form, i.e.

$$[\mathbf{A} \quad : \quad \mathbf{b}] \rightarrow [\mathbf{U} \quad : \quad \mathbf{z}]$$

(Here, $\ell_{ik} = \tilde{a}_{ik}/\tilde{a}_{kk}$, and the tildes indicate we must use the changed elements that emerge during elimination of the augmented matrix, and k denotes the current pivot row or column. Note also we commonly refer to the diagonal elements, i.e. the \tilde{a}_{kk} as the *pivots*.)

One interesting perspective on Gaussian elimination produces, as a “free” natural byproduct, another factorization of \mathbf{A} . This basic decomposition (factorization) follows directly from the fact that every elementary row operation of the form (2) above corresponds to multiplication on the left by a corresponding (lower-triangular) elementary matrix, and the inverses of these elementary matrices, as well as certain products of their inverses are easily determined. Specifically, if

$$\mathbf{L}^{(k)} \dots \mathbf{L}^{(2)} \mathbf{L}^{(1)} [\mathbf{A} \quad : \quad \mathbf{b}] = [\mathbf{U} \quad : \quad \mathbf{z}]$$

then

$$[\mathbf{A} \quad : \quad \mathbf{b}] = \underbrace{\mathbf{L}^{(1)-1} \mathbf{L}^{(2)-1} \dots \mathbf{L}^{(k)-1}}_{\mathbf{L}} [\mathbf{U} \quad : \quad \mathbf{z}]$$

Comparing the first block elements on the left and right yields:

$$\mathbf{A} = \mathbf{L} \mathbf{U} \quad (3)$$

where \mathbf{L} can be shown to be *lower* triangular with ones on the diagonal and subdiagonal elements precisely equal to the ℓ_{ik} described above, and \mathbf{U} , as already seen in Gaussian elimination, will be *upper* triangular. Furthermore, the observation about values of the ℓ_{ik} allows us to build \mathbf{L} , “on the fly,” and for no cost, as Gaussian elimination proceeds to construct \mathbf{U} . Moreover, since the subdiagonal elements of \mathbf{A} are zeroed out during the elimination, and since we never need to store elements that we know ahead of time will have a specific value, then we can in fact store the subdiagonal elements of \mathbf{L} in the space “vacated” by the zeros created during elimination. Therefore, all the information needed to recover \mathbf{L} and \mathbf{U} can be stored in the space originally occupied by \mathbf{A} . Finally, a

basic computational complexity analysis shows that the cost of Gaussian elimination, and hence of \mathbf{LU} factorization, is “only” approximately $\frac{2}{3}m^3$ flops, or about half the cost of finding the \mathbf{QR} factorization for the same \mathbf{A} .

Moreover, even incorporating row interchanges (pivoting) produces only a minor and almost equally simple to obtain variant of the factorization. Hopefully though, by now, we are also starting to view all such theoretical claims from the slightly jaundiced viewpoint of how well they actually play out when implemented for real matrices, on real computers which have real storage and execution speed limitations, and which must use real software and floating-point arithmetic. In this lab, we shall use a couple of very rudimentary **m**-file implementation of variants of \mathbf{LU} decomposition, plus MATLAB’s built-in \mathbf{LU} function to investigate some of the differences and agreements between theory and reality a bit more.

First of all, there turn out to be several mathematically equivalent ways to implement the construction of \mathbf{U} . More interestingly, and hopefully by now not unexpectedly, these different implementations, while mathematically equivalent, do not in fact produce equivalently efficient code when implemented on real computers. To help better understand this, we start with the observation that, at the first stage of Gaussian elimination, we eliminate below the diagonal in the first column of \mathbf{A} by multiplying on the left by:

$$\mathbf{L}^{(1)} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -\ell_{21} & 1 & 0 & \cdots & 0 \\ -\ell_{31} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\ell_{m1} & 0 & 0 & \cdots & 1 \end{bmatrix} \equiv \begin{bmatrix} 1 & \vdots & \mathbf{0} \\ \cdots & \cdots & \cdot \\ -\boldsymbol{\ell}^{(1)} & \vdots & \mathbf{I} \end{bmatrix}$$

where

$$\boldsymbol{\ell}^{(1)} = \begin{bmatrix} \ell_{21} \\ \ell_{31} \\ \vdots \\ \ell_{m1} \end{bmatrix} \equiv \begin{bmatrix} \tilde{a}_{21}/\tilde{a}_{11} \\ \tilde{a}_{31}/\tilde{a}_{11} \\ \vdots \\ \tilde{a}_{m1}/\tilde{a}_{11} \end{bmatrix}$$

(Note the tildes here are actually superfluous, since eliminating in the first column occurs before any other elements of \mathbf{A} have been changed yet.) If we block \mathbf{A} similarly, i.e.

$$\mathbf{A} = \begin{bmatrix} a_{11} & \vdots & \mathbf{a}^{(1)H} \\ \cdots & \cdots & \cdots \\ \mathbf{c}^{(1)} & \vdots & \mathbf{A}^{(1)} \end{bmatrix}$$

where the values of $\mathbf{a}^{(1)H}$, $\mathbf{c}^{(1)}$ and $\mathbf{A}^{(1)}$ are clear from the context, then we can easily show

$$\mathbf{U}_{work}^{(1)} = \mathbf{L}^{(1)} \mathbf{A} = \begin{bmatrix} a_{11} & \vdots & \mathbf{a}^{(1)H} \\ \cdots & \cdots & \cdots \\ \mathbf{0} & \vdots & \mathbf{A}^{(1)} - \boldsymbol{\ell}^{(1)} \mathbf{a}^{(1)H} \end{bmatrix}$$

But we also observe the computation needed to update of lower right hand “working” block in \mathbf{U}_{work} , which is formulated here as a rank one outer product, can in fact be implemented in four equivalent ways:

- (i) Element by element - i.e. with multiple *for* loops, i.e.

$$\text{for } i=2:m; \text{ for } j=2:m \text{ } \mathbf{A}(i,j) = \mathbf{A}(i,j) - \mathbf{L}(i,j)*\mathbf{A}(1,j);\text{end;end}$$
- (ii) Row by row - i.e.

$$\text{for } i=2:m; \mathbf{A}(i,2:m) = \mathbf{A}(i,2:m) - \mathbf{L}(i,1)*\mathbf{A}(1,2:m);\text{end}$$
- (iii) Column by column - i.e.

$$\text{for } j=2:m; \mathbf{A}(2:m,j) = \mathbf{A}(2:m,j) - \mathbf{A}(1,j)*\mathbf{L}(2:m,1);\text{end}$$
- (iv) By computing the result directly as a MATLAB primitive, i.e.

$$\mathbf{A}(2:m,2:m) = \mathbf{A}(2:m,2:m) - \mathbf{L}(2:m,1)*\mathbf{A}(1,2:m)$$

(Note all of these assume the values of $\text{matL}(2:m,1)$ were computed ahead of time and stored in another array.) Furthermore, similar analyses can be shown to apply to all subsequent updates that occur as we proceed through the rest of the steps of the factorization. Although we shall not pursue this issue in any greater depth here, you should realize that which of these is best in any specific application involves such subtle considerations as:

- Is the language compiled, or interpreted?
- Does the language store arrays as successive rows, or as successive columns?
- Does the language support *primitive* matrix operations, e.g. the **BLAS**?

One concern that we cannot duck, however, is the effects of floating-point arithmetic. In the case of Gaussian elimination and **LU** factorization, the single most important issue in this arena is the effect of small pivots. Specifically, we can easily see from (2) that when a_{kk} is (relatively) small, then ℓ_{kk} is (relatively) large. In finite precision, because of the need to align the decimal point before adding or subtracting, a large multiplier in

$$R_i \leftarrow R_i - \ell_{ik}R_k \quad , \quad i > k$$

will, effectively, replace R_i by $\ell_{ik}R_k$, producing a matrix (\mathbf{U}_{work}) with two almost linearly dependent rows. Such a matrix will, of course, be nearly singular, i.e. very ill-conditioned. Any subsequent calculations with this matrix will almost certainly produce garbage. So, “vanilla” Gaussian elimination is simply a non-starter for practical matrix computation!

Fortunately, there is an easy fix for this. We can simply interchange rows (or even columns), so that, whenever we eliminate in a column, the largest (in magnitude) element in the current working column is the pivot. This will ensure that all of the ℓ_{ik} have a magnitude of one or less. Moreover, when, for what ever reasons, row interchanges must be incorporated, theory also tells us that we can then produce the so-called **PLU** factorization, i.e.

$$\mathbf{P} \mathbf{A} = \mathbf{L} \mathbf{U} \tag{4}$$

In this factorization, \mathbf{P} is a permutation matrix, and, furthermore, the algorithm for finding \mathbf{P} , \mathbf{L} and \mathbf{U} should not be significantly more difficult than the algorithm for finding \mathbf{L} and \mathbf{U} without pivoting. There is one crucial caution we must mention, however. Partial pivoting can be viewed as a method for precluding the introduction of an *artificially* ill-conditioned matrix \mathbf{U} during the elimination process. Nothing, including any type of pivoting, will likely prevent the introduction of an ill-conditioned \mathbf{U} when the original matrix \mathbf{A} is well-conditioned!

In order to both study the actual flow of the calculations in this algorithm, and also to observe actual effects of finite precision arithmetic, we provide several new programs, `gepp_steps.m`, `gepp_steps_chop.m`, `lupp_chop.m`, `fwd_solve_chop.m`, and `fp_solve.m`. You should observe that, based on observations from previous labs, Gaussian elimination and LU factorization are implemented using option (iii) above, i.e. column by column updates. We have also, as in previous codes, in one of them, utilized `chop()` to simulate the effects of low-digit finite precision arithmetic, which should allow us to observe significant effects in matrices of reasonably small size. We must immediately add, however, that MATLAB already implements \mathbf{LU} decomposition with its own, built-in `lu()` command, which we shall also study to some degree. Obviously, the MATLAB version should be preferred for real applications.

Of course, in numerical analysis, the mere fact that we can perform a computation, e.g. the \mathbf{LU} decomposition, is usually of little interest unless performing that computation actually provides some tangible benefit in terms of either computational efficiency or accuracy. So, even though we can theoretically obtain \mathbf{P} , \mathbf{L} and \mathbf{U} as free byproducts of elimination, why bother? There are two answers for this question. The first is that, once we have \mathbf{P} , \mathbf{L} and \mathbf{U} we can solve

$$\mathbf{A} \mathbf{x} = \mathbf{b}$$

as

$$\begin{aligned} \mathbf{L} \mathbf{z} &= \mathbf{P} \mathbf{b} \\ \mathbf{U} \mathbf{x} &= \mathbf{z} \end{aligned} \tag{5}$$

Since each of the systems in (5) is already triangular, each of those equations can be solved in about m^2 flops, with no need for elimination, a cost negligible when dealing with systems of any real size compared to the approximately $\frac{2}{3}m^3$ flops required for elimination on the original matrix \mathbf{A} . (Also, multiplication by \mathbf{P} is in fact only a rearrangement of subscripts, not a computation!) But wait! Don't we have to pay the $\frac{2}{3}m^3$ cost anyway to find \mathbf{P} , \mathbf{L} and \mathbf{U} ? That's true. But, in fact, many practical applications occur when we must solve a given system, repeatedly, with the same left hand side (\mathbf{A}), but different right-hand sides, i.e. we must solve

$$\mathbf{A} \mathbf{x}^{(k)} = \mathbf{b}^{(k)} \quad , \quad k = 1, 2, \dots$$

For such systems, if we utilize the \mathbf{PLU} decomposition, we must pay the full $\frac{2}{3}m^3$ cost **only once**, i.e. only for $k = 1$. If we then simply save \mathbf{P} , \mathbf{L} and \mathbf{U} , all subsequent solutions cost only $2m^2$, i.e. they are essentially free!

Lastly, we close the lab with two short programs, **timing_lu.m** and **timing_solve.m**, which uses the by-now familiar **tic** and **toc** routines to confirm our earlier statements about the relative computation complexity of the different algorithms for factoring matrices, and for solving the basic systems of equations.

[This Page Intentionally Left Blank]

Name: _____

MA 3046 - Matrix Analysis
Laboratory Number 7
LU Decomposition and Numerical Accuracy

1. Login to your workstation and start MATLAB. Then, using any web browser, link to the course laboratories home page and download the programs:

gepp_steps.m, gepp_steps_chop.m, lupp_chop.m, fpsolve_chop.m
fwd_solve_chop.m, time_ge.m, and time_solve.m

plus the data file **lu_methods.mat** to your working space. In addition, make sure the files

ge_basic.m, ge_steps_chop.m, and fwd_solve_chop.m

used in earlier laboratories, are available in your work space.

2. Next, give the command

load lu_methods

and verify that you have created the matrices and vectors:

$$\mathbf{A1} = \begin{bmatrix} 2.03 & 1.00 & 9.32 & 5.25 \\ -2.01 & -0.98 & -10.5 & -3.31 \\ 6.04 & 7.47 & 4.19 & 6.72 \\ 2.72 & 4.45 & 8.46 & 8.38 \end{bmatrix}, \quad \mathbf{b1} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\mathbf{A2} = \begin{bmatrix} 4.01 & 5.90 & 5.18 \\ -1.39 & -2.31 & -1.70 \\ 7.56 & 12.0 & 9.42 \end{bmatrix} \quad \text{and} \quad \mathbf{b2} = \begin{bmatrix} 16.4 \\ -5.20 \\ 29.1 \end{bmatrix}$$

respectively.

3. Using MATLAB's **cond()**, determine the condition number of **A1**

$$\kappa(\mathbf{A1}) =$$

Then, using MATLAB's backslash command, find the solution of:

$$(\mathbf{A1}) \mathbf{x} = \mathbf{b1}$$

and record it:

$$\mathbf{x1} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

Approximately how many digits of **x1** should be correct, and why?

4. Give the command

$$\mathbf{uwork} = \mathbf{ge_basic}([\mathbf{A1} \mathbf{b1}]) ;$$

Observe the process of Gaussian elimination, without pivoting. Look especially for the appearance of small pivots. Record the final result

$$\mathbf{uwork} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

What is the condition of the portion of **uwork** corresponding to the matrix **U** from Gaussian elimination:

$$\kappa(\mathbf{U1basic}) =$$

Why is this result either reasonable or not reasonable based on the theory discussed in class?

5. Open the MATLAB text editor, and study the programs

gepp_steps.m and **gepp_steps_chop.m**

until you feel comfortable with their general logic and flow. Then give the commands:ds

global NDIGITS
NDIGITS = 3

(Note that these set all future programs that end with **_chop** to simulate a three-digit, rounding decimal machine, until a different value of NDIGITS is provided.)

6. Give the command

uwork = gepp_steps([A1 b1]) ;

Observe the process of Gaussian elimination with partial pivoting. Look especially for the appearance of small pivots. (There should not be any! Why?) Record the final result

$$\mathbf{uwork} = \left[\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \right]$$

What is the condition of the portion of **uwork** corresponding to the matrix **U** from Gaussian elimination:

$$\kappa(\mathbf{U1pp}) =$$

How do this compare with the condition number of the **U** in part 4, and how do you explain the difference?

7. Give the command

uwork = gepp_steps_chop([A1 b1]) ;

Observe the process of Gaussian elimination with partial pivoting now in a simulated three-digit machine. (There should not be any! Why?) Record the final result

$$\mathbf{uwork} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

What, if any significant differences do you see between this augmented echelon matrix and the one found using full MATLAB precision in part 6?

8. Using the **bwd_solve_chop** command from previous labs, compute the solution corresponding to the (augmented) echelon matrix in part 6 above:

$$\mathbf{x13} = \begin{bmatrix} \\ \\ \end{bmatrix}$$

and compare this result to the full MATLAB precision solution found in part 3.

9. Using MATLAB `lu` command to determine the matrices **L1**, **U1** and **P1** corresponding to **A1**:

$$\mathbf{L1} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

$$\mathbf{U1} = \begin{bmatrix} \\ \\ \\ \end{bmatrix} .$$

$$\mathbf{P1} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

To what degree do these look reasonable or not in light of applicable theory?

10. Using the matrices **L1**, **U1** and **P1** produced in part 9, compute

$$\mathbf{P1} * \mathbf{A1} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

and

$$\mathbf{L1} * \mathbf{U1} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

Why do or do not these values agree with theory?

Check your conclusion by computing

$$\mathbf{P1} * \mathbf{A1} - \mathbf{L1} * \mathbf{U1}$$

11. Using the **P1**, **L1** and **U1** determined above, and the normal MATLAB `\` command, solve the two equations

$$(\mathbf{L1})\mathbf{z} = (\mathbf{P1})\mathbf{b1}$$

$$(\mathbf{U1})\mathbf{x} = \mathbf{z} \quad ,$$

where **b1** was created in part 2 above. Solution:

$$\mathbf{z} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

and

$$\tilde{\mathbf{x}} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

Compare this result to the solution **x1** obtained from the backslash command in part 3.

12. Using your texteditor, open program **lupp_chop.m** and study it until you are comfortable that it correctly implements **L U** decomposition with partial pivoting. Also consider how other equivalent implementations might be coded. Then also look at programs **fwd_solve_chop.m**, **bwd_solve_chop.m** and **fpsolve_chop.m** until you are also comfortable that these correctly implement the solution of

$$\mathbf{A} \mathbf{x} = \mathbf{b}$$

13. Give the command:

`[L13 U13 P13] = lupp_chop(A1)`

and record the results

$$\mathbf{L13} = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

$$\mathbf{U13} = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} .$$

$$\mathbf{P13} = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

Compare these matrices to those computed in part 9. Do the two sets of matrices appear to reasonably agree in the context of the different precisions used? If not, can you see a reason why not? Does $\mathbf{L13} * \mathbf{U13}$ still reasonably approximate $\mathbf{P13} * \mathbf{A1}$?

14. Next, give the commands

$$\mathbf{A} = \mathbf{A1} \quad \text{and} \quad \mathbf{b} = \mathbf{b1}$$

and then: `fpsolve_chop`

$$\tilde{\mathbf{x}}_{13} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

Compare this answer to the one you computed in part 8 and 11. How many correct digits does this solution appear to have? What, if anything, does that suggest?

15. Using MATLAB's `cond()`, determine the condition number of **A2**

$$\kappa(\mathbf{A2}) =$$

Then, using MATLAB's backslash command, find the solution of

$$(\mathbf{A2}) \quad \mathbf{x} = \mathbf{b2}$$

$$\mathbf{x}_2 = \begin{bmatrix} \\ \end{bmatrix}$$

Approximately how many digits of \mathbf{x} should be correct, and why?

16. Give the command

uwork = gepp_steps_chop([A2 b2]) ;

Observe again the process of Gaussian elimination with partial pivoting. Look especially for the appearance of small pivots. Record the final result

$$\mathbf{uwork} = \left[\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \right]$$

Identify any qualitative differences between this augmented echelon matrix and the one you found in part 6.

Can you see any possible explanations for these differences? Specifically, if this echelon matrix still has one or small pivots, why did partial pivoting not prevent that from occurring?

17. Next, give the commands

$$\mathbf{A} = \mathbf{A2} \quad \text{and} \quad \mathbf{b} = \mathbf{b2}$$

and then: `fpsolve_chop`

$$\tilde{\mathbf{x}}_{23} = \begin{bmatrix} \\ \\ \end{bmatrix}$$

Compare this answer to the one you computed in part 15. How many correct digits does this solution appear to have? What, if anything, does that suggest?

18. Study the MATLAB **m**-files **time_ge.m** and **time_solve.m**, until you feel you understand what they are computing. Then run each (you'll have to be patient), and observe the resulting graphs. What do these indicate.

How are these results consistent with theory?